NASA-CR-191280

November 1992

UILU-ENG-92-2242
CRHC-92-24

Center for Reliable and High-Performance Computing

NCC 2-481

IN-63-CR
131805
P.20

RESOURCE CONSTRAINED DESIGN OF ARTIFICIAL NEURAL NETWORKS USING COMPARATOR NEURAL NETWORK

Benjamin W. Wah and Tanay S. Karnik

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-92-2242 CRHC-92-24	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois / 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION National Aeronautics Space Administration
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) Moffett Field CA 94035
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 7a / 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NASA NCC 2-481
8c. ADDRESS (City, State, and ZIP Code) 7b	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Resource Constrained Design of Artificial Neural Networks Using Comparator Neural Networks

12. PERSONAL AUTHOR(S)
Wah, Benjamin W. and Tanay S. Karnik

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM ____ TO ____	1992 November 19	18

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	neural networks, artificial intelligence, constrained design

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

We present a systematic design method executed under resource constraints for automating the design of artificial neural networks using the back error propagation algorithm. Our system aims at finding the best possible configuration for solving the given application with proper tradeoff between the training time and the network complexity. The design of such a system is hampered by three related problems. First, there are infinitely many possible network configurations, each may take an exceedingly long time to train; hence, it is impossible to enumerate and train all of them to completion within fixed time, space, and resource constraints. Second, expert knowledge on predicting good network configurations is heuristic in nature and is application dependent, rendering it difficult to characterize fully in the design process. A learning procedure that refines this knowledge based on examples on training neural networks for various applications is, therefore, essential. Third, the objective of the network to be designed is ill-defined, as it is based on a subjective tradeoff between the training time and the network cost. A design process that proposes alternate configurations under different cost-performance tradeoff is important. We have developed a *Design System* which schedules the available time, divided into *quanta*, for testing alternative network configurations. Its

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Resource Constrained Design of Artificial Neural Networks Using Comparator Neural Network

Benjamin W. Wah and Tanay S. Karnik
Coordinated Science Laboratory
University of Illinois
1101 W. Springfield Ave.
Urbana, IL 61801.

Abstract

We present a systematic design method executed under resource constraints for automating the design of artificial neural networks using the back error propagation algorithm. Our system aims at finding the best possible configuration for solving the given application with proper tradeoff between the training time and the network complexity. The design of such a system is hampered by three related problems. First, there are infinitely many possible network configurations, each may take an exceedingly long time to train; hence, it is impossible to enumerate and train all of them to completion within fixed time, space, and resource constraints. Second, expert knowledge on predicting good network configurations is heuristic in nature and is application dependent, rendering it difficult to characterize fully in the design process. A learning procedure that refines this knowledge based on examples on training neural networks for various applications is, therefore, essential. Third, the objective of the network to be designed is ill-defined, as it is based on a subjective tradeoff between the training time and the network cost. A design process that proposes alternate configurations under different cost-performance tradeoff is important. We have developed a *Design System* which schedules the available time, divided into *quanta*, for testing alternative network configurations. Its goal is to select/generate and test alternative network configurations in each quantum, and find the best network when time is expended. Since time is limited, a dynamic schedule that determines the network configuration to be tested in each quantum is developed. The schedule is based on relative comparison of predicted training times of alternative network configurations using *comparator network* paradigm. The *comparator network* has been trained to compare training times for a large variety of traces of *TSSE*-versus-time collected during back-propagation learning of various applications.

1. INTRODUCTION

We have developed a systematic design method executed under resource constraints for automating the design of artificial neural networks using the back error propagation algorithm. Our system aims at finding the best possible configuration for solving the given application with proper tradeoff between the training time and the network complexity. The design of such a system is hampered by three related problems. First, there are infinitely many possible network configurations, each may take an exceedingly long time to train; hence, it is impossible to enumerate and train all of them to completion within fixed time, space, and resource constraints. Second, expert knowledge on predicting good network configurations is heuristic in nature and is application dependent, rendering it difficult to characterize fully in the design process. A learning procedure that refines this knowledge based on examples on training neural networks for various applications is, therefore, essential. Third, the objective of the network to be designed is ill-defined, as it is based on a subjective tradeoff between the training time and the network cost. A design process that proposes alternate configurations under different cost-performance tradeoff is important.

In [6] , it was shown that cost and performance cannot be combined to define an objective function. Performance should be used as an objective under constrained cost. In this study, we set an upper limit on the number of hidden units in any network to constrain the cost and assume the objective function of the network to be designed is a function of current training time and predicted completion time. The behavior of *TSSE*, the total sum of squared errors, is usually oscillatory in time space and the completion time is difficult to predict. Absolute value of the prediction may be fallible, but relative values of the predicted completion times of alternative networks is a dependable basis for comparison. We assume that the user proposes a pool of possible networks before design begins, and that the system generates new ones during the design. We assume that when *TSSE*, the total sum of squared errors, reaches ε_{crit}, a user-specified critical maximum error, training in back-propagation is complete and terminated. Finally, we assume a set of training patterns for a given application is available.

2. PREVIOUS WORK

Research on artificial neural networks has been very active in the last few years [12, 16]. Technological advances made it easier to build massively parallel machines and hence models of computation are inspired by neural networks [5]. Artificial neural networks have the greatest potential in areas such as speech and image recognition, where many hypotheses are pursued in parallel, high computation rates are required, and the performance of the current best systems are far from ideal [11].

A network typically consists of a large number of simple elements that learn and collectively solve a complicated, even ill-defined, problem. The simple element is referred to as a neuron. The current state of the neuron is called its *activation*, that is a function of the inputs the neuron receives from other neurons or the external world, and the weights associated with the corresponding links. Each neuron is characterized by a number known as its *bias* and an *activation function*. Each network has one layer of input neurons and a layer of output neurons. There might be one or more hidden layers between these two layers.

In this research we consider the systematic design of feedforward networks learned by back-propagation learning algorithms [12, 16]. Such networks have been shown to apply to a large class of

Research support by National Aeronautics and Space Administration Grant NCC 2-481.

applications. In our study we used training traces generated for NetTalk, Lymphography Domain and IRIS Plants Database problems.

Although back-propagation learning is one of the most popular and powerful learning algorithms, it is a heuristic gradient descent algorithm and does not provide any guidance on how initial weights and network configurations should be chosen. Numerous enhancements on improving its gradient descent nature has been developed, but little work has been done on developing a systematic design process. Since training of a given network may or may not converge depending on the network configuration and its initial weights, and a great deal of time might have been incurred before discovering that learning does not converge, it is imperative that a poor network be found and discarded early in the design process.

There are four major methods proposed in the literature. They apply a combination of greedy search and backtracking.

Ad Hoc Method. In this method [16], one starts with a network configuration, assigns random weights to it, chooses some values of training parameters, and trains the network by applying the back-propagation algorithm. If the network does not converge, a new network is chosen. The problem with this method is that if the chosen network does not converge, the time to train this network is wasted. This may exhaust the allotted training time, leaving no promising network as a result.

Iterative Refinement Method. Fahlman [1] has developed a *Restart Method* in which learning is allowed to restart, with new random weights, when learning has failed to converge after a given number of epochs. The time reported for a trial includes the time spent before and after the restart. This method avoids the problem of nonconvergence of learning by setting a ceiling on the number of epochs to be expended on a particular network. However, it does not alter the network configuration, but only selects a new set of weights. Hence learning may not converge at all if the choice of the configuration is wrong. Moreover, the performance of the method also depends on the choice of the ceiling on epochs expended per network. A high value would result in wasting computational resources, whereas a low value may result in discarding promising configurations early in the learning process.

Evolutionary Learning. The problem of deciding whether or not a given task can be performed by a given network architecture is NP-complete [7]. There is no method for specifying a priori an appropriate network structure, neither in terms of its learning performance nor in terms of its representational comprehensibility. Many have hypothesized that the design process should be evolutionary, starting from primitive networks to more general ones [22]. For feed forward networks learned using the back-propagation algorithm, promising methods have been developed for choosing the number of hidden units [10, 15]. These approaches are broadly divided into two classes, *start big and remove* [9, 14, 17], and *start small and add* [2, 18].

Cascade-Correlation Learning [3], developed by Fahlman and Lebiere, is one of the most promising configuration generation method. Starting with a minimal network, it expands the network incrementally: hidden units are added to the network one at a time. Based on a pool of candidate hidden units, the network is trained a number of times using the training set. The hidden unit that maximizes the magnitude of the correlation between the outputs and the candidate unit's value is added to the active network and its input weights are frozen. The method does not require an educated guess of the network configuration: it can build deep networks without slowing down the performance because at any time, only one layer of weights is trained. The problem with this approach is that it represents a form of greedy search: once a hidden unit is committed to the active network, it cannot be removed even if performance is found to degrade later. A more general approach is to allow backtracking to a previous configuration once the present configuration is found to perform poorly.

Iterative Refinement with Backtracking. We developed a design method based on backtracking [20] that proved to be promising and successful (see Figure 1). The system is divided into two types of experiments: Type-I and Type-II.

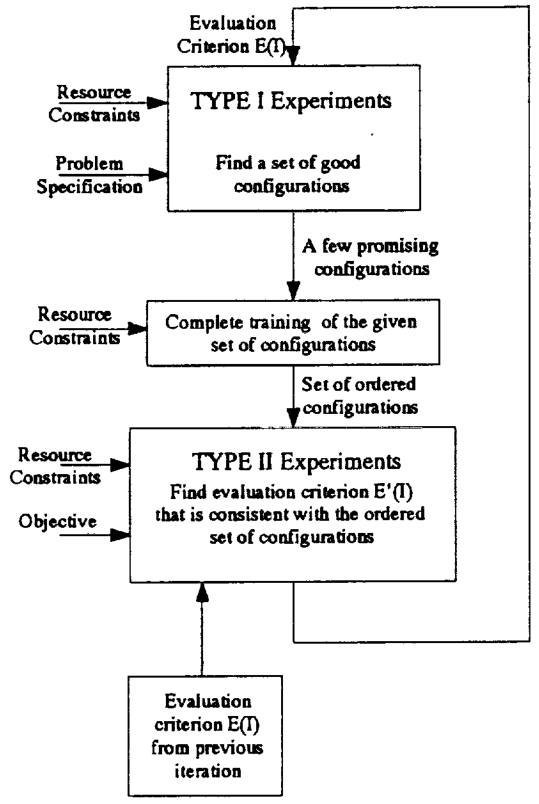
Evaluation Criterion E(I)
Resource Constraints
Problem Specification
TYPE I Experiments
Find a set of good configurations
A few promising configurations
Resource Constraints
Complete training of the given set of configurations
Set of ordered configurations
Resource Constraints
Objective
TYPE II Experiments
Find evaluation criterion E'(I) that is consistent with the ordered set of configurations
Evaluation criterion E(I) from previous iteration

Figure 1. Coupled Iterative Refinement Design Method.

Type-I experiments attempt to find a set of good configurations, given the problem specifications, the resource constraints, and an evaluation criteria for evaluating alternative configurations. These experiments are in the form of a search. In each quantum of time, the most promising network is selected and trained based on the evaluation criteria. Hence the method has a mechanism to backtrack if the current most promising network fails to perform well later. The result of Type-I experiments is a set of promising configurations, selected on the basis of the evaluation criteria.

The set of promising networks found by Type-I experiments are then fully evaluated until their *TSSE* drops below a critical error, ε_{crit}. The traces of *TSSE*-versus-time in training these networks are used as inputs to Type-II experiments, which refines and learn new evaluation criteria consistent with the traces. The new criteria learned are then used in Type-I experiments; and the process of refinement repeats.

Although the method has been successful in generating good network configurations for a number of applications, there are a few drawbacks. First, the Type-I experiments developed currently only select a promising configuration from a pool of configurations (with initial weights defined) supplied by the designers. To generalize the method, a configuration generator that proposes new networks must be included. Second, learning in Type-II experiments is based on traces of promising configurations found in Type-I experiments; such a set may be biased statistically, and the criteria learned in Type-II experiments may not generalize well to other problems. Third, the form of the evaluation criteria is not defined and it can be any function of different parameters, such as *TSSE*, *Slope* of *TSSE*, training time *T*, etc. To learn a general evaluation criteria in Type-II experiments, a variety of initial guesses and applications should be used as training inputs. Fourth, the normalization of training times and *TSSE* was done by user-supplied normalization constants: such constants again restrict the generality of the evaluation criteria learned.

With the above drawbacks in mind, we have addressed three major problems. First, our design decoupled the Type-I and Type-II experiments. We replaced the Type-II experiments by training a comparator network [8, 13, 19] to perform relative comparisons of predicted training times. Training inputs to the comparator network are drawn from a variety of applications and network sizes so that they are not biased. Second, we developed better normalization methods so that the networks are compared using their relative goodness rather than absolute goodness in the *Design System* (or Type-I experiments in the original method). This avoids the specification of normalization constants as needed before. Last, we developed a network configuration generator that generates new networks to be tested in the *Design System*. The configuration generator utilizes expert background knowledge in proposing better networks to be experimented.

3. NEURAL NETWORK DESIGN

In this section we describe our methodology for the automated design of neural networks. We first describe our *Design System*. We discuss all the essential components of the system.

3.1. Design System

The *Design System* systematically trains promising configurations for various amount of quanta of time. It selects, generates, or discards network configurations and trains promising ones until the allotted time is exhausted. The flow diagram of the system is shown in Figure 2. Descriptions of relevant terms are tabulated in Table 1. We assume that the time allowed is divided into quanta and that in each quantum limited learning is performed on one network.

At the beginning of each quantum, the Strategy Selector selects the next candidate network to be tested. If a candidate network is finished or discarded, then a new candidate network is generated. The selected or generated candidate network undergoes learning using a back propagation neural network learning system (we have developed our own Simulator) in the Strategy Applicator. The effects of learning are available readily at the end of the quantum. The Performance Database stores the *TSSE* behavior for all networks under training. The Performance Predictor then predicts using the comparator neural network the best candidate found so far on the basis of data in the Performance Database. The Strategy Selector then selects the next candidate to be tested, and the design cycle continues until the allotted time is exhausted. In the rest of this section, we describe research issues considered in each component of the *Design System*.

The **Strategy Selector** selects or generates a candidate network to be tested in the next quantum of time based on static and dynamic strategies stored in the system. Since limited amount of neural network learning can be performed in each quantum, the proper strategy to be used cannot be determined a priori. The major issues involved are the selection of the momentum, the learning rate, the guidance function,

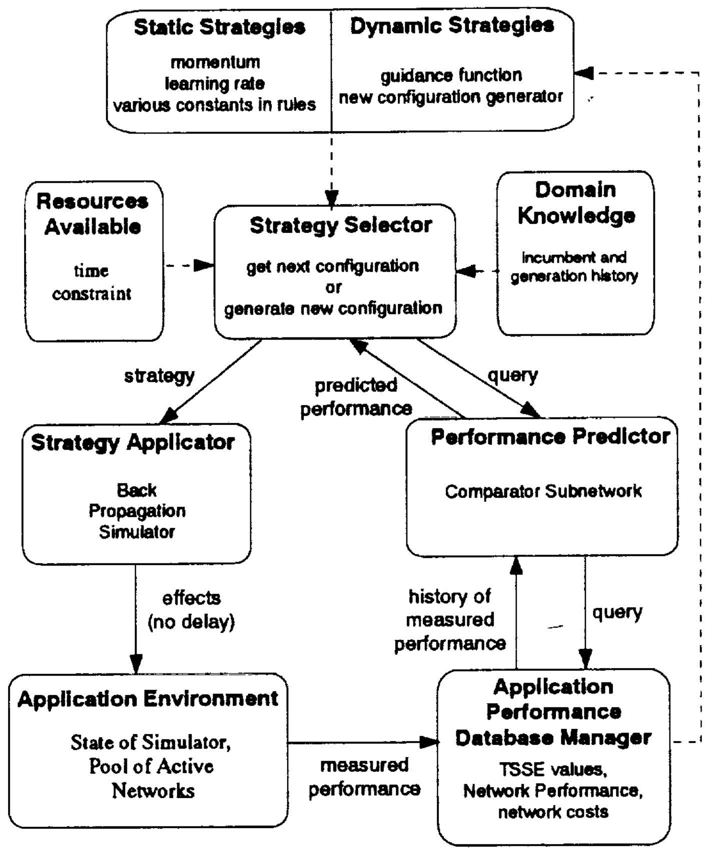

Figure 2. Artificial Neural Network Design System.

and the method for generating a new candidate network. We fix the momentum and the learning rate in our neural network simulation system based on previous results in this area. The guidance function is expected to be a function of the goodness value, epochs expended on a network, and quanta remaining. Note that the guidance strategy selected is a combination of backtracking and greedy search: backtracking is useful when the time remaining is large, whereas a greedy search is more promising when time is running out. The candidate generator that we adopt at this time is based on evolutionary techniques, such as *cascade correlation* [3].

The **Strategy Applicator** trains the candidate network selected for a limited number of epochs in a quantum of time. We have developed our own back-propagation neural network simulator as opposed to the preexisting simulators because simulating a pool of networks consumes a large amount of memory and we needed to have a control over memory usage. Also we wanted the simulator to stop and report the performance at the end of each quantum. The tradeoff to be considered here is the size of the quantum: if the quantum is too large, then fewer networks will be examined within a time limit, but more accurate performance results can be assessed at the end of a quantum.

The **Application Performance Database** maintains the history of performance of all candidate networks under consideration, including measures such as the history of goodness values of candidate networks, average time taken for testing a candidate network, performance after every test, the costs of networks experimented., and the list of *TSSE* values.

Table 1. Terms used in the Design System.

Term	Description
Candidate	The network under consideration as specified by its configuration, initial weights, current learning rate, and momentum.
Candidate Pool	Pool of candidates divided into active list, finished list, and discarded list.
Cost	$-f\,(N_{units}\uparrow, N_{weights}\uparrow, N_{biases}\uparrow)$
Goodness Value	Measure for selecting the best candidate network if the process were to be stopped. Same as performance.
Guidance Strategy	Measure for selecting the candidate network for further learning in the next quantum if the experiments were to be continued.
TSSE	Total sum of squared errors between the actual outputs produced by the candidate and the corresponding training outputs.
ε_{crit}	Maximum critical error so that learning of a network is considered complete if its *TSSE* is less than ε_{crit}.
Incumbent	The candidate network with the best goodness value at a given time.
Basis Candidate	A candidate whose *TSSE* is reduced below a predetermined multiple of ε_{crit} and the slope of TSSE-time plot at the time considered is negative. Used as a basis for generating new candidate.
Discarded Network	A candidate network that has undergone at least one test and whose goodness value is more than two standard deviations below the average goodness value of the active candidate networks.
Best Network	The incumbent when time is expended.

The **Performance Predictor** determines the best candidate if the design process were to be stopped at any time. We assume that this prediction is based on computing a goodness value for each network and selecting one with the best goodness value. We assume that the goodness value of a candidate network is some unknown function of the network's predicted completion time and total time spent in the neural network simulator. We realize this function using a comparator network paradigm which is explained in implementation details.

4. DESIGN ISSUES AND APPROACH

In this section, we discuss our research on addressing the issues in the *Design System*. Systematic methods are developed for resolving these issues and for overcoming the problems associated with our design method. We discuss each of these issues along with our solutions.

4.1. Candidate-Network Generator

In our *Design System*, we assume that a pool of active candidate neural network configurations are provided by the users initially and that the system generates new configurations when needed. The pool should consist of at least one candidate network specification. As discussed in Section 2, in order for the system to be able to generate networks in an incremental fashion, we apply an evolutionary learning technique in generating new configurations.

Our evolutionary learning technique to generate new candidate networks is as follows:

1) The generator accepts the current *candidate pool* and *comparator goodness function* as inputs.
2) Select a candidate randomly out of the top one fourth goodness valued candidates. It is considered as *basis candidate*. If it is not tested at all then *incumbent* is chosen as the *basis candidate*.
3) List initial assertions for the production rulebase system for all the active candidates, basis and incumbent.
4) Consider the candidates whose *TSSE* value is reducing and close to ε_{crit} as *well tested candidates*.
5) Record the relationship between candidates. Two candidates are related if they have the same number of layers and the difference between the number of hidden units is less than half of *maximum perturbations* allowed. We call these candidates to be in the vicinity of each other.
6) Record the difference in performance in the related candidates. Locate the maximum performance candidate in the vicinity of the *basis candidate*. If there is such candidate then consider that candidate as the *basis candidate*.
7) Check whether the *basis candidate* is in the top two third of the *well tested candidates*. Choose *incumbent* as the *basis* otherwise.
8) Find the greatest performance improvement *transform* of the *basis candidate*. Here *transform* means the difference in the network configuration.
9) Check whether the *transform* is valid, i.e. it does not violate cost constraints, it has not been tried already and yields a feasible new network. Explore an unexplored direction otherwise, i.e. randomly perturb the configuration.
10) If this perturbation of the *basis candidate* is not going to exceed the limit, randomly add/subtract hidden units to generate a new configuration in the vicinity of the *basis candidate*. If a unit is added, then the weights of the new network are set to the same values for the corresponding links of the earlier basis candidate network. The weights of the links to and from the added unit(s) are chosen randomly. Similarly, when a unit is removed, the unit and its corresponding links are removed, and the rest of the weights retain their values.
11) If the vicinity of the *basis candidate* is exhausted or there is no *transform* feasible under cost constraints, then the same configuration with new initial set of random weights is generated as a new candidate.
12) Since we need to use the epochs expended in a candidate network in computing its goodness value, we assume that the epochs expended on a new candidate be a fraction of the epochs expended on the basis network. The particular fraction used is currently set to half of the epochs expended on basis candidate.

4.2. Comparator Network

Goodness measure of a candidate is an ill-defined concept. It should be a measure of selecting the most promising network. In the domain of artificial neural network design, it should be inveresely proportional to the actual completion time. It can be a function of *TSSE*, *Slope* of *TSSE*-versus-time curve, its *Second derivative*, moving averages or statistics of these quantities, *epochs* expended, and/or *predicted completion time*. The form of the function is also ill-defined. Hence absolute measure of goodness is difficult to compute. The problem of selecting the best candidate from a pool of candidates involves comparison among the candidates. A relative goodness measure instead of absolute goodness measure suffices the needs. In [13], Mehra and Wah have presented a novel learning architecture, called *comparator neural network* for relative comparison of completion times of candidate networks. We briefly describe the architecture in this section.

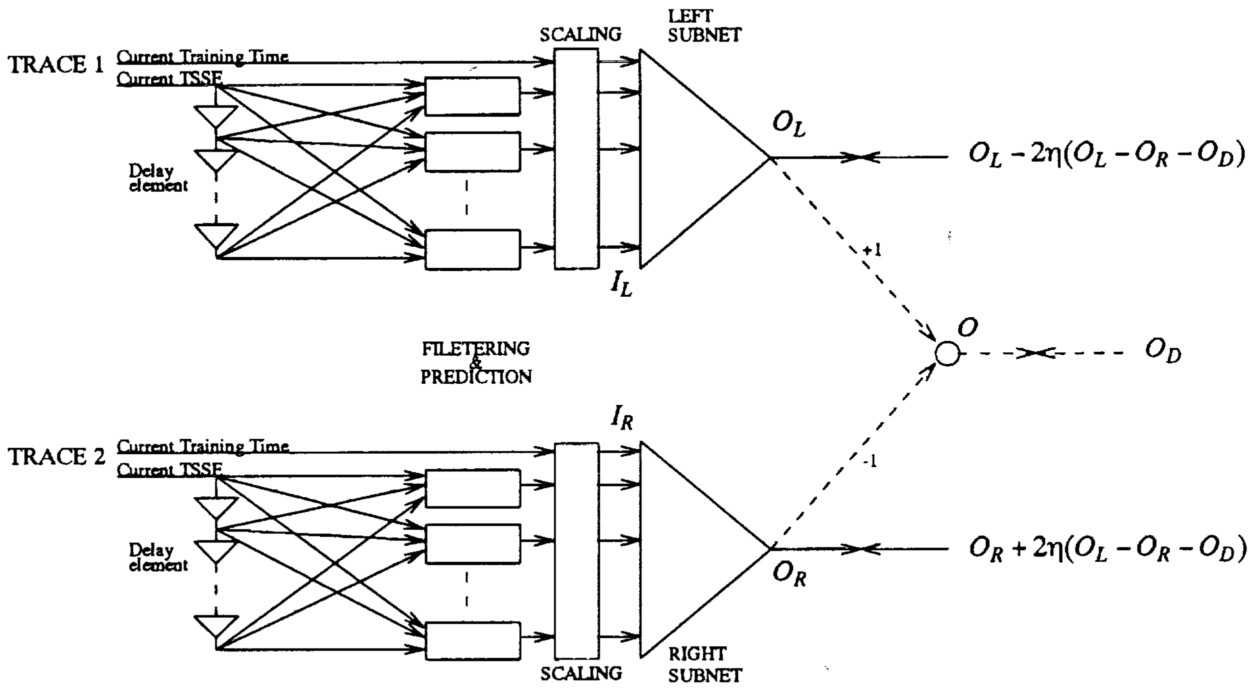

Figure 3. Comparator Neural Network.

4.2.1. Description

We assume that the goodness value of a candidate is a nonlinear function of the candidate's current training time and predicted completion time. As the form of the function is ill-defined, we realize this function by a comparator neural network.

A comparator neural network is shown in Fig. 3. It consists of two identical subnets, viz. *left-subnet*(L) and *right-subnet*(R). Two input vectors to be compared, I_L and I_R are applied to each subnet. The network has to learn to compare and choose the more promising input between the two applied inputs. We train the network to learn the difference in desired outputs.

Let the desired output difference be O_D. The inputs I_L and I_R produce outputs O_L and O_R respectively. As the two subnets are identical it is same function which maps input to output in both subnets. Let $O = O_L - O_R$. The objective then becomes to minimize the cost function:

$$E = (O - O_D)^2$$

A derivation similar to back-propagation algorithm leads us to the teaching outputs to both the subnets as follows:

$$\frac{\partial E}{\partial O_L} = \frac{\partial E}{\partial O}\frac{\partial O}{\partial O_L} = 2(O-O_D)$$

$$\Delta O_L = -\eta\frac{\partial E}{\partial O_L} = -2\eta(O-O_D)$$

$$O_L^{teach} = O_L-2\eta(O-O_D) = O_L-2\eta(O_L-O_R-O_D)$$

Similarly,

$$\frac{\partial E}{\partial O_R} = \frac{\partial E}{\partial O}\frac{\partial O}{\partial O_R} = -2(O-O_D)$$

$$\Delta O_R = -\eta\frac{\partial E}{\partial O_R} = 2\eta(O-O_D)$$

$$O_R^{teach} = O_R+2\eta(O-O_D) = O_R+2\eta(O_L-O_R-O_D)$$

Teach outputs are computed for every pattern presentation from the actual desired outputs and current subnet outputs and conventional back-propagation learning is performed. In order to train the subnets identically, we apply the inputs commutatively, i.e. each pair of inputs is presented as I_L-I_R and also I_R-I_L.

4.2.2. Trace Generation

Inputs to the comparator network are derived from the traces of *TSSE*-versus-time for a variety of networks for each problem. The networks used must represent an unbiased sample of possible networks used in the *Design System*. Note that the same network may be initialized with different initial weights.

We have considered only 2-layer networks. The upper bound on the total number of hidden units is the number of training patterns, and the lower bound is 1. We have considered three benchmark problems for these experiments, viz., NetTalk, IRIS Plants Database and Lymphography Domain. For each of these problems, we trained at least 40 different networks with our back-propagation simulator. We chose 5 converged networks and 5 nonconverged networks (upto 3000 epochs) for training, and 1 converged network and 2 nonconverged networks for testing, for each benchmark problem leading to traces of 39 networks.

4.2.3. Pattern Generation

The *TSSE*-versus-time traces of all these networks were sampled loglinearly and linearly at approximately 20 different points. Each of the sampled traces was filtered using 4 kinds of Butterworth filters and two fitting techniques, linear and exponential. Completion times were predicted from the filtered and fitted data. The estimated values and the actual completion time were scaled linearly and loglinearly to lie between [0–1] interval for sigmoid neurons. Empirically, we have observed that linear sampling and linear scaling yields the best results. Scaled completion training time for nonconverged traces was set to 3.0 heuristically. These 8 predicted values with the current sampling epochs (scaled) were fed to comparator network as input patterns. Each converged network is compared against each other converged and nonconverged network, for the same benchmark problem. Nonconverged networks are not compared among themselves.

As the values of current sampling time, actual completion time and predicted completion times are scaled linearly, there is a strong dependence on scaling denominator, maximum training time, $t_{\max}$. In all our experiments we have set $t_{\max}$ to be 10000 epochs, but this value may have to be increased for mapping to [0–1] interval. Hence we inject 10% noise in $t_{\max}$. We randomly select a value between 10000 and 100000 for $t_{\max}$ with 10% probability.

4.2.4. Training and Testing

All input-output patterns for all 3 benchmark problems yield 58304 training comparisons and 3120 testing comparisons. The testing network traces are entirely different from training traces. The accuracy of the comparator network is same as number of times the sign of the difference in actual outputs of the subnets, $O_L - O_R$, is consistent with the sign of the difference in actual completion times. Extensive experiments were performed by varying hidden units, learnrate, momentum of the comparator subnets, and sampling and scaling methods.

4.2.5. Implementation in the Design System

Choice of the comparator network involved the tradeoff between training and testing accuracy, and size of the network. Currently, we have selected a comparator network with 9–15–1 configuration with learnrate 0.2 and momentum 0.5 trained on linearly sampled and scaled data. It yielded 75% accuracy both on training and testing data after 664 training epochs.

4.2.5.1. Conversion to a Function

The trained comparator neural network is in the form of a weightsfile. As the patterns applied commutatively, both the subnets are identical and they realize the same function. This function learned by a subnet, produces a value in [0–1] interval proportional to the completion time of a network and hence it behaves opposite of the sought *goodness function* in our *Design System*. We define *goodness value* to be (1.0 - O_L). As the network has already been trained and rebuilding and simulating a subnet is an unnecessary overhead on the *Design System*, we built a C-routine which functions exactly as the simulated *left-subnet* from the stored weights. An automated facility to build a C-routine from the dumped weightsfile is developed for this purpose.

4.2.5.2. Incremental Scaling Denominator

We have addressed the problem of removing the dependence on scaling denominator, $t_{\max}$, in *Design System* also. First, we start with a predetermined value(10000) for $t_{\max}$. After each quantum, we check whether the number of epochs expended on the current network is in the 10% vicinity of $t_{\max}$. If such a case exists, $t_{\max}$ is increased by 10% and goodness values of all active candidate networks is updated. A possible disadvantage may be that the number of goodness values to be computed at the end of such a quantum is equal to the number of active networks in the *Design System*.

4.3. Parameter Initialization

Various parameters have to initialized in the *Design System*. We expect the user to set some of these parameters and the others are set by the system internally depending on the application problem specification. We will discuss in this section how they are set.

4.3.1. User Defined Parameters

Like any ANN simulator, we expect the user to define application problem specifications and the stopping criterion, ε_{crit}. The user must specify at least one probable network configuration in the initial pool of active networks. A flag (*FEED-F*) needs to be set if direct links from input units to output units are desired. In addition to these parameters, we expect the user to set the *number of active candidates* to be considered at the beginning of any quantum. This setting can be based on the total time available and problem complexity. The user has to specify the number of times the pruning heuristic to be called for discarding poor performing candidates and generating new candidates. In candidate generation, we need the user to set the limit on number of perturbations of the same *basis network* and define the vicinity to ε_{crit} for *well-tested candidates.*

The *pre_learning_cycles* represents the number of epochs that need to be expended on each network in the initial candidate pool in order to extract the trend in the *TSSE*-versus-time curve to predict completion time. This also helps in avoiding the possible initial transients in the learning curve. This parameter is usually chosen to be 50 so that we can filter and fit the *TSSE*-versus-time curve reliably.

4.3.2. Internally Set Parameters

The *epochs_per_test* represents the number of training patterns that will be iterated in testing the candidate network once. Ideally, all the training patterns should be cycled once in each epoch. However, if the application is large, such as NetTalk, this time may be prohibitive. We are planning to determine a suitable function for this parameter. The parametric form of this function is

$$epochs\_per\_test = f(T\uparrow, N_{pat}\downarrow, M\downarrow)$$

where T is the total time available to the *Design System*, N_{pat} is the number of entries in the input-output pattern set, and M is a machine-dependent parameter that accounts for the load and speed of the application environment. It can be set to the actual time required for executing one iteration of the back-propagation algorithm on the network with the maximum cost. The above function would give a tradeoff between the number of epochs per test and the complexity of the application. Currently we have implemented a function which depends only on the number of entries in the pattern set, N_{pat}. Please note that *epochs_per-test* can be set to a fractional value also.

$$epochs\_per\_test = \begin{cases} trunc\,(7-.005\,N_{pat}) & \text{if } N_{pat} < 1000 \\ 1 & otherwise \end{cases}$$

The *average_quantum_duration* represents the size of a quantum of time, during which a single network is tested. If the quantum size is too large, then too few networks will be tested in a fixed amount of time, although more certainty can be attributed to the goodness values at the end of a test. We measure the time to build and simulate a 2-layer network with hidden units equal to the number of input-output patterns for one test. As this is the maximum allowed network for the given application problem under cost constraints and the actual candidate networks are far less complex, we heuristically set *average_quantum_duration* to one fourth the above measured time.

In generating a new neural network configuration, a number of parameters need to be determined. In order to limit the complexity of the network, we set the maximum layers to be 5 (*maximum_layers*), and limit the total number of hidden units (*maximum_hidden_units*) by the number of input-output pattern set. In generating a new network configuration, we need to determine a **vicinity criterion** that identifies whether the new configuration is similar to the ones already generated. Two networks are considered in vicinity of each other if they have the same number of layers and difference in the number of hidden units is less than half of maximum perturbations allowed. Finally, a **pruning criterion** on when a network is to be discarded from the active list needs to be specified. Whenever the pruning heuristic is activated, we discard the candidates whose *goodness values* are two standard deviations less than the mean of *goodness values* of all active candidates.

4.4. Output of the Design System

As discussed in Section 1, the objective function of our design process is not well defined. Users usually determine subjectively whether a solution meets their requirements after it has been proposed, and when an adequate solution is found, they may wish to find another one that surpasses it. The design process can be repeated with different initial values for parameters and different initial candidate pool. Again, this method reduces our lack of knowledge on the objective to a systematic search of different objectives that meet the requirements.

The system stores the configuration and initial weights of all the networks built in the complete run. Currently, the system outputs the list of all candidates with their *TSSE*, *training times*, *goodness* and *guidance values* at the end. If a network converges in the specified time limit, the system quits immediately outputting current weights of that network and the list sorted by goodness values. All the networks can be rebuilt and trained to completion to validate the consistency of comparator neural network.

5. FUTURE WORK

5.1. Experimental Results

We plan to compare the above approach based on evolutionary learning with one based on a greedy approach. In the latter method, a new network configuration is generated based on perturbation of the network with the best goodness value. Note that a method based on backtracking is better when the time allowed is sufficiently large, whereas a greedy search, such as *cascade correlation*, performs better under a tight time constraint. A proper tradeoff between the two approaches remains an issue to be studied in this research.

5.2. Stopping Criteria

Currently, a network is considered to be converged if *TSSE* falls below ε_{crit}. Another way of determining convergence is implemented in *cascade correlation*. For classification application problems, output of an output neuron is considered to be 1 if it lies in the interval [0.6–1] and 0 if it lies in the interval [0–0.4]. In each test, number of output neurons producing wrong output is computed. If there is no such output neuron for all the patterns in the test, then the network is claimed to be converged. Our system monitors the wrong output bits in every test and with a slight modification, this stopping criterion can be implemented in the *Design System.*

5.3. Trace storage

Currently, *TSSE* values after every test are stored in the *performance database*. For a large number of active networks and high time limit, the storage of all *TSSE* values can overload memory. There is a need of a method to sample the *TSSE* values stored and store only a limited set. Initial values in the learning curve are important to extract the downward trend and current values are essential to determine the closeness to ε_{crit}. Also the sampling can be loglinear or linear. In our future work, we would like to address this problem carefully. In the current system, we limit the number of active candidates and store all the trace values.

The *comparator network* used for predicting completion time has been trained over the pattern set generated using the complete trace from the beginning of the learning. If a new method of storing traces is implemented, then the *comparator network* should be trained and tested again.

5.4. Issues in Generation

The candidate networks simulated can have upto 5 layers. Each layer is linked to the next higher layer. There is an additional provision for linking input units to output units. The system does not allow any additional links. There may be a need to have links from any lower unit to any higher unit. A network with all lower units linked to all higher units is difficult to parallelize but learns faster. Please note that in a 2-layer network, our system provides all such links and it has been shown that two layers are sufficient to learn a given input-output mapping using back-propagation algorithm [10].

Generation of new candidate by perturbing the *basis candidate* involves change in the number of hidden units in one of the hidden layers. There is no provision to add(remove) a layer to(from) the configuration. The production rulebase system for generation is certainly a time consuming program. Generation is required only when the initial pool of candidates needs more networks or a candidate is discarded in the system run. As it is not called frequently, it is not an overhead, but if new pruning heuristics are implemented then generation may be a significant time overhead. A simplified version with reduced complexity of candidate generation is desired.

5.5. Learning Guidance Function

In our current system, we have a greedy approach to guidance strategy. A guidance function of the following form may be used.

$$guidance = f(goodness \uparrow, epoch \downarrow, quanta\_remaining)$$

This function is heuristic in nature and has to be learned using TEACHER architecture applying population based learning strategies [21]. The best network at any stage in the course of training may get stuck in a local minimum later. Hence, the function should penalize the network if the epochs expended were large. In contrast, the guidance function is not monotonic with respect to *quanta_remaining*. When *quanta_remaining* is low, it may be necessary to focus experimentation on the better networks found so far; whereas if the time remaining is large, the resource scheduler may schedule less promising networks to be trained.

6. ADDITIONAL UTILITIES

The following utilities are provided in the current system but not used as they are redundant in our framework. The system does utilize these utilities and they can be activated in later use to develop a more efficient *Design System.*

6.1. Normalization of Goodness Values

If the *goodness measure* is based on various application-dependent factors such as *TSSE*, *Slope* of *TSSE*-versus-time curve, its *Second derivative*, moving averages or statistics of these quantities, *epochs* expended then the quantities that may affect a goodness value include $\varepsilon_{\max}$, the maximum *TSSE* value, $t_{\max}$, the maximum number of epochs allowed for training any network for this application, and $C_{\max}$, the maximum cost of any network configuration to be used. As it is important that a goodness function learned for one application is not restricted to this application, these application-dependent normalization constants must be selected properly.

In finding $\varepsilon_{\max}$, we assume that the value of any output unit varies between 0 to 1 and that the desired outputs are binary. Hence, the average difference between the target output and the actual output per output unit per pattern is 0.5. The sum of average *TSSE* values for all output units and all training patterns is, therefore, $(N_{pattern} \times N_{output} \times 0.5^2)$, where $N_{pattern}$ and N_{output} are the number of training patterns and output units, respectively. Note that 0.5 is squared because we are measuring the sum of squared errors. From experience, we assume that $\varepsilon_{\max}$ is a constant multiple of the above quantity; that is,

$$\varepsilon_{\max} = 1.5\, N_{pattern}\, N_{output}\, 0.5^2$$

To normalize the cost of a network, we note that for any neural network application, the maximum number of hidden units required in a 2-layer network (with one hidden layer and one output layer) is equal to the number of training patterns. Hence, the cost of this "maximum" network can be used to normalize the costs of networks considered in the *Design System.*

The selection of $t_{\max}$, the maximum number of epochs for normalization, is more difficult, as it is difficult to predict the maximum time to stop learning in a neural network. To overcome this problem, we can use the incremental scaling denominator method in *comparator network paradigm.*

6.2. Moving Averages

The *application performance database* can store window as well as auto-regressive moving averages of different quantities such as *TSSE, Slope of TSSE, Second Derivative of TSSE.* Our system provides all utilities to calculate these moving averages.

For window averages, the system expects the user to specify length of different windows for above quantities. In using auto-regressive moving averages, a number of constants may need to be set. For instance, *k* in the following formula needs to be determined.

$$ARMA^{new}_{TSSE} = k\, TSSE_{current} + (1-k) ARMA^{old}_{TSSE}$$

The proper constant to be used will be determined experimentally.

6.3. Learnrate-Momentum Update

In [4] , Frazini has proposed a method to update learnrate according to the progress of the network towards convergence. If the *TSSE* is reducing(increasing) for successive tests then the learnrate is increased(reduced) by a small fraction and kept unchanged otherwise. We have implemented this approach to update learnrate and momentum for individual candidate networks. The user has to specify minimum number of cycles the network should go through before this update is activated. This parameter can be same as *pre_learning_cycles.* Currently, we have bypassed this utility to reduce the complexity of our *Design System.*

[1] S. E. Fahlman, "Faster-Learning Variations on Back-Propagation: An Empirical Study," *Proc. Connectionist Models Summer School*, pp. 38-51, Morgan Kaufmann, Palo Alto, CA, 1988.

[2] S. E. Fahlman and Christian Lebiere, "The Cascade-Correlation Learning Architecture," in *Advances in Neural Information Processing Systems 2*, ed. D. S. Touretzky, pp. 524-532, Morgan

Kaufmann, San Mateo, 1990.

[3] S. E. Fahlman and Christian Lebiere, The Cascade-Correlation Learning Architecture (CMU-CS-90-100), School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA, Feb. 1990.

[4] M. A. Frazini, "Speech Recognition with Back Propagation," *Proc. 9th Annual Conf. IEEE Engineering in Medicine and Biology Society*, pp. 1702-3, IEEE, 1987.

[5] G. E. Hinton, "Connectionist Learning Procedures," *Artificial Intelligence*, vol. 40, pp. 185-234, Elsevier Science Pub., New York, 1989.

[6] A. Ieumwananonthachai, A. Aizawa, S. R. Schwartz, B. W. Wah, and J. C. Yan, "Intelligent Process Mapping Through Systematic Improvement of Heuristics," *J. of Parallel and Distributed Computing*, vol. 15, pp. 118-142, Academic Press, June 1992.

[7] S. Judd, "Learning in Networks is Hard," *Proc. IEEE First Conf. on Neural Networks*, vol. 2, pp. 685-692, 1987.

[8] T. Karnik, B. W. Wah, and P. Mehra, *Learning to Predict Relative Completion Times using Comparator Neural Networks*, 1992 (submitted for review).

[9] J. K. Kruschke, "Creating Local and Distrtibuted Bottlenecks in Hidden Layers of Back-Propagation Networks," *Proc. Connectionist Models Summer School*, pp. 120-126, Morgan Kaufmann, Palo Alto, CA, 1988.

[10] S. Y. Kung and J. N. Hwang, "An Algebraic Projection Analysis for Optimal Hidden Units Size and Learning Rates in Back-Propagation Learning," *Proc. Int'l. Conf. Neural Networks*, vol. I, pp. 363-370, IEEE, 1988.

[11] R. P. Lippmann, "An Introduction to Computing with Neural Nets," *Accoustics, Speech and Signal Processing Magazine*, vol. 4, no. 2, pp. 4-22, IEEE, April 1987.

[12] J. L McClelland and D. E. Rumelhart, *Explorations in Parallel Distributed Processing: A Handbook of Models, Programs, and Exercises,* Vol. 3, MIT Press, Cambridge, MA, 1988.

[13] P. Mehra and B. W. Wah, "Adaptive Load-Balancing Strategies for Distributed Systems," *Proc. 2nd Int'l Conf. on Systems Integration*, pp. 666-675, IEEE Computer Society, Morristown, NJ, June 1992.

[14] M. C. Mozer and P. Smolensky, "Using Relevance to Reduce Network Size Automatically," *Connection Science*, vol. 1, no. 1, pp. 3-16, Carfax Pub. Co., Oxfordshire, United Kingdom, 1989.

[15] P. Rujan and M. Marchand, "Learning by Minimizing Resources in Neural Networks," *Complex Systems*, vol. 3, pp. 229-241, Complex Systems Pub. Inc., Champaign, IL, 1989.

[16] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, ed. D. E. Rumelhart, J. L McClelland and the PDP Research Group, vol. 1, pp. 318-362, MIT Press, Cambridge, MA, 1986.

[17] J. Sietsma and R. J. F. Dow, "Neural Net Pruning--Why and How?," *Int'l Joint Conf. on Neural Networks*, vol. 1, pp. 325-333, IEEE, 1988.

[18] G. Smith, *Back Propagation with Dynamic Topology and Simple Activation Functions*, Tech. Rep. TR 90-12, School of Information Science and Technology, The Flinders Univ. of South Australia, Adelaide, 1990.

[19] G. Tesauro and T. J. Sejnowski, "A Parallel Network that Learns to Play Backgammon," *Artificial Intelligence*, vol. 39, pp. 357-390, Elsevier Science Pub., New York, 1989.

[20] B. W. Wah and H. Kriplani, "Resource Constrained Design of Artificial Neural Networks," *Proc. Int'l Joint Conf. on Neural Networks*, vol. III, pp. 269-279, IEEE, June 1990.

[21] B. W. Wah, "Population-Based Learning: A New Method for Learning from Examples under Resource Constraints," *Trans. on Knowledge and Data Engineering*, vol. 4, no. 5, pp. 454-474, IEEE, Oct. 1992.

[22] G. Weiss, *Combining Neural and Evolutionary Learning: Aspects and Approaches (Tech. report FKI-132-90)*, Technische Universitaat Muenchen, Institut fuer Informatik, Muenchen 2, West Germany, 1990.